Symbolic Worst Case Execution Times*

Ernst Althaus[1], Sebastian Altmeyer[2], and Rouven Naujoks[3]

[1] Johannes-Gutenberg-Universität Mainz and Max-Planck-Institut für Informatik
ernst.althaus@uni-mainz.de
[2] Saarland University
altmeyer@cs.uni-saarland.de
[3] Max-Planck-Institut für Informatik
naujoks@mpi-inf.mpg.de

Abstract. In *immediate* or *hard real-time systems* the correctness of an operation depends not only upon its logical correctness, but also on the time in which it is computed. In such systems, it is imperative that operations are performed within a given deadline because missing this deadline constitutes the failure of the complete system. Such systems include medical systems, flight control systems and other systems whose failure in responding punctually results in a high economical loss or even in the loss of human lives.

These systems are usually analyzed in a sequence of steps in which first, a so-called control flow graph (CFG) is constructed that represents possible program flows. Furthermore, bounds on the time necessary to execute small code blocks are computed along with bounds on the number of possible executions of the program loops. Depending on the type of the analysis, these loop bounds can either be numerical values or symbolic variables, corresponding to inputs given for instance by a user or by sensors. In the last step, in such a CFG the weight of a longest path with respect to the loop bounds is computed, reflecting a bound on the worst case execution time.

In this paper, we will show how to compute such symbolic longest path weights in CFGs of software with a rather regular structure like software developed for hard real-time systems. We will present the first algorithm that is capable of computing such paths in time polynomial in the size of both the input *and* the output. Our approach replaces the application of integer linear programming solvers in the case of purely numerical loop bounds. Furthermore, it improves upon the speed and accuracy of existing approaches in the case of symbolic bounds.

1 Introduction

Immediate real-time systems require tasks to finish in time. To guarantee the timeliness of such systems, upper bounds on the worst case execution times of the their tasks have to be computed. To be useful in practice, such an analysis must be *sound* to ensure the reliability of the guarantee, *precise* to increase the chance of proving the satisfiability of the timing requirements, and *efficient*, to make them useful in industrial practice.

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, www.avacs.org).

The high complexity of modern processors and modern embedded software hampers the analysis to achieve all three properties at once. Exhaustive measurement, for instance, may be sound and precise but is infeasible for realistically sized programs. Simple end-to-end measurements are easy to derive, but are possibly unsound. We refer to [12] for a general overview of methods for the worst-case execution-time problem and to [10],[7] presenting the results of challenges in which state-of-the-art tools are compared. A general approach to achieve all three properties is to represent a program P that has to be analyzed as a so-called *control flow graph* in which a node corresponds to a *basic code block* B of P, i.e. to a maximal sequence of consecutive operations of P such that only B's first operation is the target of a jump operation in P and such that the last operation in B is a jump directive. The edges in such a graph represent the source/target relationship of the jumps in P. In a set of analyses, upper bounds on the running times of the basic blocks and upper bounds on the number of loop iterations are determined. The execution time bound of the program is then given by the weight of a longest path P in the CFG that respects the given loop bounds. The step of the timing analysis that computes such a path is usually referred to as *path analysis*.

Timing analyses treating bounds on the maximal number of loop iterations as numerical values are referred to as *numeric* or *traditional timing analyses*. The drawback of such an analysis is that bounds on loop iterations must be known statically, i.e. during design time. Some systems need guarantees for timely reactions which are not absolute, but dependent on inputs of for example a user or a sensor. In such cases, traditional timing analysis offers only two possibilities. Either one provides constant bounds for the unknown variables or one starts a new analysis each time the task is used with different inputs. While the first option endangers precision, the second may be an unacceptable increase in the analysis time. *Parametric timing analyses* circumvents this problem. Instead of computing numeric bounds valid for specific variable assignments only, parametric analysis derives symbolic formulas representing upper bounds on the task's execution times.

Traditionally, the computation of a longest path in a CFG is done by formulating the problem as an integer program. While in the case of numeric timing analyses, the corresponding integer program is linear and can be solved by an ILP-solver, the integer program in the presence of symbolic bounds on the loop iterations is in general non-linear, which makes it necessary to relax the non-linear constraints, leading to a loss of accuracy (see 2.2 for a more detailed discussion).

Even the traditional path analysis is NP-hard since by setting all loop bounds to one, the problem reduces to the longest path problem in graphs. In this paper, we discuss a new and purely combinatorial approach for the symbolic path analysis and for arbitrary loop bounds, exploiting the rather regular structure of software written for such time critical systems. Such software must be clearly structured, avoiding for instance constructs like gotos, to make the code verifiable. For CFGs arising from such code, we present an approach for the path analysis, which solves two major problems of the previous approaches: in stark contrast to the exponential running time of the ILP approach, our algorithm has a running time that is polynomial in the size of both the input and the output. Note that in case of numeric path analysis, only one path is reported. Furthermore, in the case of symbolic analyses, our algorithm avoids any relaxation of

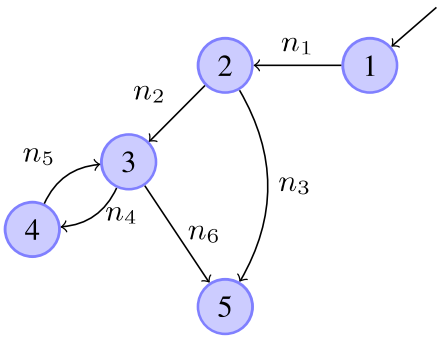
non-linear constraints, leading to potentially better bounds. We also show that for a certain type of CFGs the size of the output is minimal.

2 Related Work

In this section we briefly discuss two state-of-the-art approaches for path analysis which make use of an ILP formulation. For a detailed discussion we refer to [3,8,11]. Note, that in some cases it is convenient to assign the runtime of a basic blocks to the outgoing edges of the corresponding node in the CFG. We also restrict the discussion to the computation of a longest path from a source node s with in-degree 0 to some sink node t with out-degree 0.

2.1 Numeric Path Analysis

For the longest path computation usually a technique called *implicit path enumeration* (see [8,11]) is used. Each edge e_i is assigned a variable n_i called the *traversal count* denoting how often e_i is traversed by a control flow. Since a control flow enters a nodes v exactly as often as it leaves it, the sum of the traversal counts of edges entering v must equal the sum of the counts of the edges leaving v. For the source node and the target node we have the special rule, that the sum of traversal counts of incident edges must equal 1 since each program flow enters s and leaves t exactly once. In a third class of inequalities, the number of times a loop is traversed is limited to the number of times it is entered multiplied by its loop bound. The objective function of this ILP is then to maximize the sum over the costs of the basic blocks times the traversal counts of edges entering them (see 1 for an example).



$$\max : \sum_i \left(\sum_{j:\ n_j \text{ enters } i} c_i n_j \right)$$
$$n_1 = 1;$$
$$n_1 = n_2 + n_3;$$
$$n_2 + n_5 = n_4 + n_6;$$
$$n_4 = n_5;$$
$$n_4 \leq b \cdot n_2;$$
$$n_3 + n_6 = 1;$$

Fig. 1. A control flow graph and the corresponding ILP formulation. The running time of the basic block i is denoted as c_i. It is transfered to the incoming edges of i. b denotes the loop bound of the loop $\{3, 4\}$. Equalities ensure the flow-balance, i.e. a node is entered exactly as many times as it is left. The inequality $n_4 \leq b \cdot n_2$ ensures that the loop is traversed (i.e. edge n_4 is used) only b times the number of times it is entered (i.e. edge n_2 is used). In case of numberic loop bounds, the inequality is linear. If the loop bound is symbolic, i.e. b is a variable and we are seeking for all paths which are longest of at least one b, the inequality is non-linear and has to be relaxed by finding an upper bound on n_2.

2.2 Parametric Path Analysis

In the approach given in [9] and [3] for the parametric path analysis basically the same ILP construction as in the previous discussion is used. The difference is now that the loop bounds used in this formulation are now variables, yielding non-linear constraints – in the example 1, there is for instance the non-linear constraint $n_4 \leq b \cdot n_2$ if the loop bound b is symbolic. In order to solve such an integer program, these non-linearities must be relaxed by bounding the variables in the ILP by constants. For instance, is it easy to verify that in our example, the constraint $n_4 \leq b \cdot n_2$ can be replaced by the constraint $n_4 \leq b$, since $n_2 \leq 1$. Once, all non-linear constraints are relaxed we are left with a parametric ILP, i.e. with an ILP containing symbolic constants, which can be solved by a parametric ILP-solver as proposed by Feautrier [5] using symbolic versions of the simplex [4] and the cutting plane algorithm [6]. The two big disadvantages of this approach are that such parametric ILP-solvers are much slower than existing ILP solvers and that the method yields a loss in accuracy by the relaxation of the constraints.

3 Longest Paths in Singleton-Loop-Graphs

So far we have treated the concept of a program loop as given. But to discuss our algorithm we have to formally define what a program loop corresponds to in the CFG. Considering nested loops in a piece of code, one can observe, that any two points can be reached from each other by a possible control flow, leading to the definition that a loop is a strongly connected component in the CFG. The sub-loops of a loop L are then just the strongly connected components of L after removing the node over which L is entered. Note that this also means that the sub-loops of a loop depend on the node over which L is entered.

Assuming wlog. that such a program was written in a high level language like c, the constructs causing the control flow to branch are *for*, *repeat-until*, and *while-do* loops with *continue* and *break* statements, *if-then-else* constructs, *function calls* and *goto* directives. Recall that we want to compute upper bounds on the worst case execution times of programs developed for immediate real-time systems. As discussed before such programs typically have a rather regular structure. That is, usually the use of directives like *goto*, *break* and *continue* is avoided. If a program now does not contain *goto* directives, our definition of a loop coincides with the intutive concept of a loop. Moreover, there is a unique node over which the loop is entered, which we call the entry node of the loop, and thus, each loop must have uniquely defined sub-loops. The same holds recursively for all sub-loops. We call graphs with this property singleton-loop graphs. Beside a CFG and upper bounds for the runtime of the basic blocks, we assume that we are given loop bounds for all induced sub-loops of the CFG, which can either be explicit numerical values or symbols. In this section we will first describe the algorithm for singleton-loop graphs, achieving a polynomial running time in the size of the input and the output. In the following section, we will describe a slight variation of this algorithm for certain graphs, which we call *while-loop-graphs* in which we additionally disallow break statements. For this type of graph, we will show that we can even guarantee, that the output is of minimal size.

3.1 Preliminaries

Definition 1 (Loop). *Given a directed graph $G = (V, E)$, we call a strongly connected component $S = (V_S, E_S)$ in G with $|E_S| > 0$, a* loop *of G. We denote by* $\mathrm{loops}(G)$*, the set of all loops of G.*

Definition 2 (Entry Node). *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$ of G, we call $e \in V_L$ such that there exists an edge (u, e) in $\delta_G^+(V \setminus V_L)$ an* entry node *of L.*

Definition 3 (Singleton-Loop). *A loop L in a graph G is called a* singleton-loop *if L has exactly one entry node e. For the unique entry node of a singleton loop L, we write $\mathscr{E}(L)$.*

Definition 4 (Sub-loops). *Given a loop $L = (V_L, E_L)$, we define*

$$\mathrm{sloops}(L) := \bigcup_{v \text{ is entry node of } V_L} \mathrm{loops}(G_v)$$

where G_v is the subgraph induced by $V_L \setminus \{v\}$.

Definition 5 (Induced Sub-loops). *Given a loop $L = (V, E)$, we call the recursively defined set*

$$\mathrm{iloops}(L) := \{L\} \cup \Big(\bigcup_{L_s \in \mathrm{sloops}(L)} \mathrm{iloops}(L_s) \Big)$$

the set of induced sub-loops *of L. For a graph G, we extend the definition of* iloops *to graphs:*

$$\mathrm{iloops}(G) := \bigcup_{L_s \in \mathrm{loops}(G)} \mathrm{iloops}(L_s)$$

We call a graph G a *singleton-loop graph* if each induced sub-loop of G is a singleton-loop. For such a graph, we write $\mathscr{E}(G) := \{\mathscr{E}(L) \mid L \in \mathrm{iloops}(G)\}$ to denote the set of entry nodes of all induced sub-loops in G.

Definition 6 (Portal Nodes, Transit Edges). *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$ in G, we call $\mathcal{T}(L) := \delta_G^+(V_L)$ the set of* transit edges *of L, i.e. the edges, leaving the loop L, and $\mathcal{P}(L) := \{p \in V_L \mid \exists\ (p, v) \in \mathcal{T}(L)\}$ the set of* portal nodes *of L. A portal node is thus a source node of a transit edge.*

Note that there is a one-to-one correspondence between singleton-loops and their entry nodes, which justifies the following definition.

Definition 7 (Loop-Bound Function). *Given a singleton-loop graph $G = (V, E)$, we call a function $\mathrm{b} \colon \mathscr{E}(G) \to \mathbb{N} \cup \{+\infty\}$ a* loop-bound function *for G.*

Now we have to classify the valid paths, i.e. the paths that respect the loop-bound conditions. If for a loop L, a loop-bound of $\mathrm{b}(\mathscr{E}(L))$ is given, we say that an execution path is not allowed to enter L and iterate on L more than $\mathrm{b}(\mathscr{E}(L))$ times, before the path leaves L again.

Definition 8 (Valid Path). *Given a singleton-loop graph $G = (V, E)$, two nodes $s, t \in V$ and a loop-bound function* b *for G, we call a path $P := s \rightsquigarrow t$ a* valid path *if for all $L := (V_L, E_L) \in \mathrm{iloops}(G)$ and for all sub-paths $(\mathscr{E}(L), v_0, v_1, \ldots, v_k)$ of P with $v_i \in V_L$, the sub-path $(\mathscr{E}(L), v_0, v_1, \ldots, v_{k-1})$ contains at most $\mathrm{b}(\mathscr{E}(L))$ times the node $\mathscr{E}(L)$.*

In the following, we write $\mathrm{lps}(G, s, t)$ for a longest valid path from a node s to a node t and for a $\overline{\mathrm{lps}(G, s, t)}$ to denote the longest valid path from s to t, that contains t exactly once. Most of the times, we will limit the discussion to the task of computing just the path weights for sake of simplicity. Note, that this is not a real limitation, as the algorithm can easily be extended to also cope with the problem of reporting the paths as well. Furthermore, we will assume that for each $v \in V$ there is a path from s to t containing v. All other nodes can be removed by a preprocessing step in time $O(|V| + |E|)$. Note that the resulting graph has at least $|V| - 1$ edges.

3.2 The Algorithm

Let us recall that a problem instance is given by a singleton-loop graph $G = (V, E)$, a source node $s \in V$, an edge weight function $w\colon E \mapsto \mathbb{N}$ and a loop-bound function $\mathrm{b}\colon \mathscr{E}(G) \to \mathbb{N} \cup \{+\infty\}$. As we implicitly compute the single source all destination problem we do not specify t in the algorithm. Since from now on, we will only talk about singleton-loop graphs, we will only write loops instead of singleton-loops.

The algorithm uses the following observation. Assume the longest path traverses a loop L. It does so by traversing the longest cycle starting at the entry node in L for $b(\mathscr{E}(L)) - 1$ times and then traversing via a longest path to one of its portal nodes. Hence, our algorithm first computes for each loop L its longest loop and the longest paths to its portals in a recursive manner. This is done by splitting the entry node into one having the outgoing edge, the other having the incoming edges. The longest paths to the portal nodes do not change and the longest cycle is the longest path from the copy of the entry node with the outgoing edges to the one with the incoming edges. Then we contract the loops to single nodes where the transit edges (v, w) over portal node v, are assigned the weight that corresponds to the longest path that cycles through the entry node $b(\mathscr{E}(L)) - 1$ times, then goes to v and uses the transit edge (v, w). Finally, notice that in order to return the path itself, we have to store the path that we used to reach the entry node, denoted by $\overline{\mathrm{lps}}(G, s, \mathscr{E}(L))$, which is not the longest path to $\mathscr{E}(L)$ as this is allowed to traverse a cycle in L $b(\mathscr{E}(L)) - 1$ times. In pseudo-code our algorithm looks as follows.

$LPS(G, s) :=$

1. Identify the loops $(L_j)_{j \in \{1,\ldots,l\}}$ of G by computing the strongly connected components.
2. For each $L_j = (V_{L_j}, E_{L_j})$:
   (a) modify L_j by replacing $\mathscr{E}(L_j)$ by two nodes $\mathscr{E}_{\mathrm{out}}$ and $\mathscr{E}_{\mathrm{in}}$ and by replacing all incoming edges $(v, \mathscr{E}(L_j))$ by edges $(v, \mathscr{E}_{\mathrm{in}})$ and all outgoing edges $(\mathscr{E}(L), v)$ by edges $(\mathscr{E}_{\mathrm{out}}, v)$
   (b) call $LPS(L_j, \mathscr{E}_{\mathrm{out}})$

(c) For all $v \in V_{L_j}$ we set

$$\mathrm{lps}(G, \mathscr{E}(L_j), v) := (\mathrm{b}(\mathscr{E}(L_j)) - 1) \cdot \mathrm{lps}(L_j, \mathscr{E}_{\mathrm{out}}, \mathscr{E}_{\mathrm{in}}) + \mathrm{lps}(L_j, \mathscr{E}_{\mathrm{out}}, v),$$

(d) replace L_j in G by a single node r_j and add an edge (r_j, x) for each $(p, x) \in \mathcal{T}(L_j)$ with appropriate weights, namely: $w(r_j, x) := lps(G, \mathscr{E}(L_j), p) + w(p, x))$. Add an edge (v, r_j) for each $(v, \mathscr{E}(L_j)) \in E$ and set $w(v, r_j) := w(v, \mathscr{E}(L_j))$

3. We call the altered graph the condensed graph G'. It is a DAG, thus we can easily determine the longest paths.
4. Compute the longest path weights to nodes within the loops: Replace the nodes r_j again by the corresponding loops and set for each $L_j = (V_{L_j}, E_{V_j})$ and for all $v \in V_{L_j}$:

$$\mathrm{lps}(G, s, v) := \overline{\mathrm{lps}}(G, s, \mathscr{E}(L_j)) + \mathrm{lps}(L_j, \mathscr{E}(L_j), v)$$

So far, we haven't discussed, how the $\overline{\mathrm{lps}}(G, s, \mathscr{E}(L_j))$ in step 4 are computed. Note, that $\mathscr{E}(L_j)$ corresponds to a contraction node c in the condensed graph G'. When we compute the longest path weight from s to c, we set $\overline{\mathrm{lps}}(G', s, c) := \max_{v \in \mathrm{inc}(c)} \mathrm{lps}(G', s, v) + w(v, c)$, where $inc(c)$ denotes the set of nodes v such that there is an edge (v, c) from v to c.

Running Time - Numeric Bounds. Let us first analyze the algorithm's running time $T(|V|, |E|)$ for the case in which all loop-bounds are numeric values. In step 1), the strongly connected components of G are computed, which can be done in $O(|V| + |E|)$ time by depth-first search. Step 2a) can be computed in $O(\deg(\mathscr{E}(L_j)))$ time. In step 2b), the algorithm is called recursively which takes $T(|V_{L_j}| + 1, |E_{L_j}|)$ time. The weight updates in 2c) can be performed in $O(|V_{L_j}|)$ and the updates in 2d) in $O(|\mathcal{T}(L_j)|)$ time. It is folklore, that the computation of longest path weights in a DAG, as done in step 3), takes no more than $O(|V| + |E|)$ time. Finally, step 4 can be done in $O(|V|)$. Thus, without the recursive calls, we have a linear running time of $O(|V| + |E|)$. Note that the recursion depth of our algorithm is bounded by $|V|$, as each node is split at most once. Furthermore, the edge sets of the sub-loops are disjoint. Although nodes are split, we can argue that the total number of nodes in a certain recursion depth is bounded by $2|V|$ as follows: Let $V^{out} = \{v \in V \mid v \text{ has at least } 1 \text{ outgoing edge}\}$ and $V^{in} = \{v \in V \mid v \text{ has at least } 1 \text{ incoming edge}\}$. Then $\sum_{L \in \mathrm{sloops}(G)} |V_{\bar{L}}^{out}| \leq |V^{out}|$ and $\sum_{L \in \mathrm{sloops}(G)} |V_{\bar{L}}^{in}| \leq |V^{in}|$, where $V_{\bar{L}}$ is the set of nodes of L after splitting the entry node. Thus, in total we have

$$T(|V|, |E|) = O(|V| \cdot (|V^{out}| + |V^{in}| + |E|)) = O(|V| \cdot |E|)$$

Running Time - Symbolic Bounds. In the presence of symbolic loop-bounds we have to change our algorithm slightly. Notice, that an ordinary longest path algorithm starts with a lower bound on the length of the longest path and iteratively increases the path length if a longer path is found. In the presence of symbolic loop bounds, paths may become incomparable, i.e. if one path has length 4, the other $2 \cdot b$ for a symbolic loop

bound b. Hence, instead of a unique longest path, we now have to consider for each target node a set of paths to that node, that may be longest for a particular choice of the symbolic loop bounds (see Figure 2 for an example). When concatenating two paths we now have to concatenate all pairs of paths. Since the operations on the path weights include multiplications and additions, they can be represented as polynomials over the symbolic loop-bounds. Clearly, we aim at getting all possible path weights that are maximal for at least one choice for the symbolic loop-bound parameters. On the down side, testing whether a path weight is maximum for some choice (or instantiation) of the parameters seems to be non trivial. A compromise is to keep all paths with weights that are not dominated by another weight (i.e. all coefficients in the weight polynomial are at least as big as the coefficients in the other weight polynomial) to keep the solution set sparse in practice, which can be implemented very efficiently. Furthermore, as we will see later, for a some certain class of CFGs this step is necessary and sufficient to compute a minimal number of paths. For a problem instance $I = (G, s, t)$, consisting of a graph, a source node s and a destination node t, we denote by $\mathcal{D}(I)$ (or short $\mathcal{D}(s, t)$, if G can be deduced from the context) the set of longest path weights from s to t computed by our algorithm. The property of $\mathcal{D}(I)$, that its elements are pairwise non-dominating can be achieved by eliminating dominated elements after the execution of step 2c. We write $\mathrm{slbs}(I)$ for the number of symbolic loop-bounds of a problem instance $I = (G, s, t)$ and $\mathrm{lbs}(I) := \mathrm{lbs}(G) := |\mathrm{iloops}(G)|$ for the number of induced loops of G.

Theorem 1. *The algorithm's running time is polynomial in the input size and in the size of the output.*

Proof. First note that the running time only changes for the parts of the algorithm in which calculations on path weights are performed, namely the parts 2c), 2d) and 4). We will restrict this proof to the operations involved in step 2c), since the number of operations involved in 2c) is certainly not smaller than the ones in 2d) and 4).

Let us first count the number of operations on weight polynomials. Consider a longest path P from the source node s to the destination node t. Let $\mathrm{lps}(u, v)$ denote the longest path weights, computed by the algorithm for the longest paths from node u to node v, then for each loop $L = (V_L, E_L) \in \mathrm{loops}(G)$ that is traversed by P, we have $|\mathrm{lps}(\mathscr{E}(L), p_L)| \leq |\mathrm{lps}(s, p_L)|$ for $p_L \in \mathcal{P}(L)$ over which P leaves L again. Furthermore, for each $p_L \in portals(L)$ we have $O(|\mathrm{lps}(\mathscr{E}(L), \mathscr{E}(L))| \cdot |\mathrm{lps}(\mathscr{E}(L), p_L)|)$ operations, since the addition involves the addition of all pairs of weights in $\mathrm{lps}(\mathscr{E}(L), \mathscr{E}(L))$ and in $\mathrm{lps}(\mathscr{E}(L), p_L)$. Since L is strongly connected, $|\mathrm{lps}(\mathscr{E}(L), \mathscr{E}(L))| \leq |\mathrm{lps}(\mathscr{E}(L), p_L)|$. Thus the number of operations is bounded by $|\mathrm{lps}(\mathscr{E}(L), p_L)|^2 \leq |\mathrm{lps}(s, p_L)|^2$. Since each node in V_L can be a portal node of L, the total number of operations on polynomials occurring on the first recursion level is bounded by $\sum_{v \in V} |lps(s, v)|^2 \leq \left(\sum_{v \in V} |lps(s, v)|\right)^2$. But, since $\sum_{v \in V} |lps(s, v)|$ is just the number of path weights, reported by the algorithm, the number of operations on polynomials is polynomial in the number of reported path weights. Note that each weight has a unique representation and that all operations on the weight polynomials can be carried out in time polynomial in the size of these polynomials.

What is left to show is, that the weight polynomials computed for the nodes in the input graph have a size polynomially bounded by the size of the weight polynomials

that are reported by our algorithm, that is the weight polynomials of the longest paths from the source node s to the sink node t. We will use a structural induction over the input graph G to prove so. If G contains no loops, the claim is true since G must be a DAG and therefore, all computed longest path weights are just constants. So, let us assume that G contains loops. By induction hypothesis, the claim holds now for each problem instance $(L, \mathscr{E}_{\text{out}}, p)$ where L is a loop of G, where the entry node of L is split into the nodes $\mathscr{E}_{\text{out}}$ and $\mathscr{E}_{\text{in}}$ and where p is an arbitrary portal node of L. But then the claim is also true for $(L, \mathscr{E}(L), p)$ what can be seen as follows: Recall that a longest path weight from $\mathscr{E}(L)$ to p is given by the equation

$$\text{lps}(G, \mathscr{E}(L), p) = (\text{b}(\mathscr{E}(L)) - 1) \cdot \text{lps}(L, \mathscr{E}_{\text{out}}, \mathscr{E}_{\text{in}}) + \text{lps}(L, \mathscr{E}_{\text{out}}, p)$$

for some path weights $\text{lps}(L, \mathscr{E}_{\text{out}}, \mathscr{E}_{\text{in}})$ and $\text{lps}(L, \mathscr{E}_{\text{out}}, p)$. But then, $\text{lps}(G, \mathscr{E}(L), p)$ is as least as large as the maximum of the sizes of $\text{lps}(L, \mathscr{E}_{\text{out}}, \mathscr{E}_{\text{in}})$ and of $\text{lps}(L, \mathscr{E}_{\text{out}}, p)$ as each term in $\text{lps}(L, \mathscr{E}_{\text{out}}, \mathscr{E}_{\text{in}})$ appears with a multiple of $\text{b}(\mathscr{E}(L))$, $\text{lps}(L, \mathscr{E}_{\text{out}}, p)$ does not contain $\text{b}(\mathscr{E}(L))$ and each term in $\text{lps}(L, \mathscr{E}_{\text{out}}, p)$ can eliminate only terms that are not multiplied with $\text{b}(\mathscr{E}(L))$. The last thing we have to show now is, that the claim holds for (G', s, t), where again G' denotes the condensed graph. We compute the longest path weights in the directed acyclic graph G' by the recurrence $\text{lps}(G, s, u) = \max_{v \in \text{inc}(u)} (\text{lps}(G, s, v) + w(v, u))$ starting with $u := t$. Consider now a weight polynomial $P = \text{lps}(G, s, v) + w(v, u)$. Since we consider the condensed graph G', $w(v, u)$ is a polynomial containing only variables associated with the loop that in turn is associated with the node v (in the case that v is not a condensed node, $w(v, u)$ is just a constant). Thus, except for the constant terms, P contains at least as many terms as there are in $\text{lps}(G, s, v)$ or in $w(v, u)$, which completes the proof.

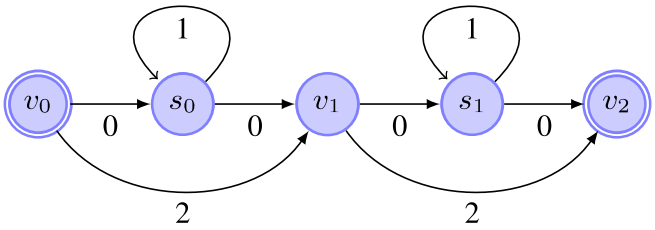


Fig. 2. The different weights for the longest paths from v_0 to v_2 are 4, $2 + \text{b}(s_0)$, $2 + \text{b}(s_1)$ and $\text{b}(s_0) + \text{b}(s_1)$

Correctness. Now, we will show, that our algorithm indeed computes the weight of a longest valid path $\text{lps}(G, s, t)$ from a source vertex s to a destination vertex t. In the following, when talking about paths we always mean valid paths. Again we will assume that G is a singleton-loop graph with weight function $w\colon E \mapsto \mathbb{N}$ and that we are given a loop-bound function $\text{b}\colon \mathscr{E}(G) \to \mathbb{N} \cup \{+\infty\}$. We will show the claim by induction over the recursion-level of the algorithm. If we assume that G contains no loops, G must be a directed acyclic graph and thus, our algorithm is correct. So, now assume that G contains loops. The induction hypothesis tells us now that for all recursive calls of our algorithm, we obtain correct results. Let $p := s \rightsquigarrow t$ be a longest path in G. Let us assume w.l.o.g. that p shares at least one node with a sub-loop of G, i.e. for some

$L := (V_L, E_L) \in \text{sloops}(G)$: $\mathscr{E}(L) \in V_L$. Thus p can be written as $p = s \rightsquigarrow p' \rightsquigarrow t$ with $p' = (\mathscr{E}(L) = v_0, v_1, \ldots, v_k)$ such that $v_i \in V_L$ and k is maximal. Since any sub-path of a longest path must be again a longest path between its starting- and end-node (with respect to the validity), we have that $w(p') = \text{lps}(G, \mathscr{E}(L), v_k)$. Consider now the condensed graph G' obtained by replacing loop L by a node r as described in the algorithm. Then the path $s \rightsquigarrow r \rightarrow v_k \rightsquigarrow t$ is valid and has weight $w(p)$. Therefore, $w(\text{lps}(G, s, t)) \leq w(\text{lps}(G', s, t))$. On the other hand, $w(\text{lps}(G, s, t))$ cannot be strictly less than $w(\text{lps}(G', s, t))$, because otherwise there would be a path in G' with weight strictly greater than a longest path in G, which also would not traverse L, since the weights of these paths are unequal. But this would mean, that there is also a path in G – just bypassing L – with the weight $w(\text{lps}(G', s, t))$, which leads to a contradiction.

What is left to show is, that our algorithm computes correct values for $\text{lps}(G, \mathscr{E}(L), v_k)$. Let $p = (\mathscr{E}(L) =: v_0, v_1, \ldots, v_k)$ with $v_i \in V_L$ be a longest path. We can assume that p contains exactly $\text{b}(\mathscr{E}(L))$ (respectively $\text{b}(\mathscr{E}(L))+1$ if $v_k = \mathscr{E}(L)$) times the node $\mathscr{E}(L)$, otherwise we could extend the path by the path $v_k \rightsquigarrow \mathscr{E}(L) \rightsquigarrow v_k$ without violating validity. Now each sub-path p' of p with $p' = \mathscr{E}(L) \rightsquigarrow \mathscr{E}(L)$ must have the same weight, since otherwise, by replacing the lower weight sub-path by the corresponding higher weight sub-path, we could obtain a path with higher weights. Thus, we can assume that there exists a longest path $\mathscr{E}(L) \overset{p'}{\rightsquigarrow} \mathscr{E}(L) \overset{p'}{\rightsquigarrow} \cdots \overset{p'}{\rightsquigarrow} \mathscr{E}(L) \overset{p''}{\rightsquigarrow} v_k$ with weight $(\text{b}(\mathscr{E}(L))-1) \cdot w(p') + w(p'')$. Since p' and p'' must be longest paths, we are left to show that our algorithm computes the weights $\overline{\text{lps}}(G, \mathscr{E}(L), \mathscr{E}(L))$ and $\overline{\text{lps}}(G, \mathscr{E}(L), v_k)$ correctly. But this follows directly by the way we alter the loop L, i.e. by splitting the entry node of L into the two nodes $\mathscr{E}_{\text{out}}$ and $\mathscr{E}_{\text{in}}$. Since L was a loop, every node in L is reachable from $\mathscr{E}_{\text{out}}$. by induction hypothesis the algorithm now computes recursively the right values, where obviously $w(\text{lps}(\overline{L}, \mathscr{E}_{\text{out}}, \mathscr{E}_{\text{in}})) = w(\overline{\text{lps}}(G, e, e))$.

3.3 While-Loop-Graphs

Even though our algorithm runs in time polynomial in both the input size and the output size, we haven't discussed so far, how large the output size can get. One can show that there is a problem instance such that the number of paths that any correct algorithm has to report is $2^{\text{slbs}(G)}$. Unfortunately, we can only show that our algorithm produces outputs such that the number of reported paths is not larger than $2^{2^{\text{lbs}(G)}}$ and even worse, there is a problem instance such that the minimal output size is 2 while our algorithm reports $2^{2^{\text{lbs}(G)-1}}$ paths. On the other hand, experiments have shown that in practice the number of reported paths is quite small. For proofs of these claims and for some experimental results we want to refer to Appendix A, respectively to [1]. Unfortunately, a precise discussion of the conducted experiments goes beyond the scope of this paper.

In this section we will discuss our algorithm on a certain type of graphs with the additional property that each induced sub-loop has exactly one portal node which coincides with the entry node of the loop. The motivation for considering such CFGs is that if additionaly to goto directives we also avoid break statements, the CFG corresponds to a while-program. Since a while–loop is entered and left only via its loop header, we can assume wlog. that the corresponding CFG exhibits this special property. The reason why we haven't considered this case before is of practical nature. Converting a program into a while–program can change its worst case running time. On the other hand, most

of the CFGs that we have considered in experiments, consisted mostly of such loops. Thus, the ideas presented in this section, can also lead to a significant reduction in the output size in practice.

Thus, we assume for now that the CFG, has the additional property that $\forall L \in \text{iloops}(G)\colon\ \mathcal{P}(L) = \{\mathscr{E}(L)\}$. We now modify our algorithm as follows. Since the entry node of a loop always coincides with its only portal node, we have in step 2c): $\text{lps}(G, \mathscr{E}(L_j), \mathscr{E}(L_j)) = \text{b}(\mathscr{E}(L_j)) \cdot \text{lps}(L_j, \mathscr{E}_{\text{out}}, \mathscr{E}_{\text{in}})$ and can thus avoid the addition operation. In Lemma 1 we will show, that for this class of graphs, the reported path weight polynomials are small. Then we will state in Lemma 2 that also the worst case number of distinct longest paths is significantly smaller in such graphs. Finally, we show in Theorem 2 that the modified algorithm reports a minimal number of path weights.

Lemma 1. *For any problem instance I, $\mathcal{D}(I)$ contains only weights with at most* $\text{slbs}(G) + 1$ *terms with non-zero coefficients.*

Proof. First observe that by the construction of our algorithm, all weights are indeed polynomials over the symbolic loop-bounds. We will now show, that any such polynomial consists of at most $\text{slbs} + 1$ terms, by first proving several claims about the possible structure of such terms and by concluding from that, that there cannot be more than $\text{slbs} + 1$ such terms in the polynomial. The claims to show are:

1. Any path in G enters the loops of G in a unique order, i.e. there don't exist two paths P_1, P_2 in G, such that there exist loops L_1, L_2 of G such that the paths P_1 and P_2 can be written as $P_1 = (\dots, \mathscr{E}(L_1), \dots, \mathscr{E}(L_2), \dots)$ and $P_2 = (\dots, \mathscr{E}(L_2), \dots, \mathscr{E}(L_1), \dots)$.
2. Any term in the weight polynomial contains only variables associated with the induced sub-loops of one $L \in \text{loops}(G)$.
3. Let $L \in \text{loops}(G)$ and $L' \in \text{iloops}(L)$. Furthermore, let $(L_i)_{i=1\dots k}$ be the sequence of induced sub-loops of L such that $L = L_1$, $L' = L_k$ and such that L_{i+1} is a sub-loop of L_i for all $i \in \{1, \dots, k-1\}$. If a term T in the weight polynomial contains the variable associated with L', then T also contains all variables associated with the loops L_i.

We now prove the first claim, by assuming that such two paths P_1 and P_2 exist. This implies that there are two paths (sub-paths of P_1 and P_2) from $\mathscr{E}(L_1)$ to $\mathscr{E}(L_2)$ and vice versa. Since L_1 and L_2 are loops of G, L_1 and L_2 are strongly connected. But we have just seen that the two nodes $\mathscr{E}(L_1)$, $\mathscr{E}(L_2)$ are also connected, which directly implies that L_1 and L_2 can't be strongly connected components of G, which leads to a contradiction.

For the second claim, suppose there exists a term, containing variables associated with induced sub-loops of two different loops L_1 and L_2 of G. By construction of our algorithm, multiplication (and thus, addition of a variable to a term) only occurs, if $L_1 \in \text{iloops}(L_2)$ or if $L_2 \in \text{iloops}(L_1)$. Thus, L_1 must be a subgraph of L_2 or vice versa and hence, L_1 and L_2 can only be both loops of G, if $L_1 = L_2$.

For the last claim, let us assume otherwise, and let there be a $j \in \{1, \dots, k-1\}$ such that the associated variable of L_j is not contained in T but also such that

L_{j+1} contributes its variable to T. By the construction of our algorithm, the variable associated with L_{j+1} was added to T, after splitting $\mathscr{E}(L_j)$ and identifying L_{j+1} as a strongly connected component of this altered graph. But splitting the node $\mathscr{E}(L_j)$ also involves the multiplication of the weight polynomial with the variable associated with L_j, which contradicts the fact, that this variable is not contained in T.

Now let us count the number of possible terms in a longest path weight, that contain at least one loop-bound variable. For each loop L of G we have a possible set of terms. For each term in such a set, we know that only variables associated with induced sub-loops of the corresponding loop of G are contained in it. Since a variable must be contained in such a term whenever the variable of a sub-loop is contained in the term, there are exactly $\mathrm{slbs}(L)$ possible terms per set. Thus in total, the number of possible terms, that contain at least one variable is bounded by $\sum_{L \in \mathrm{loops}(G)} \mathrm{slbs}(L) = \mathrm{slbs}(G)$. Together with the fact, that there is only one term, containing no variables, this completes the proof.

Lemma 2. *For any problem instance $I = (G, s, t)$, there are at most 2^{slbs} distinct longest path weights.*

Proof. We will give only the idea of the proof and assume for sake of simplicity that all loop bounds are symbolic. A detailed proof can be found in [1]. Basically, we prove the claim by structural induction. If G is a DAG, we are done. Thus, assume that G contains loops. Note that in the following we will implicitly assume that all mentioned paths have maximal length. The main insight is, that two paths with distinct weights must traverse the induced sub-loops of G in a different way. Consider now an s-t path P in G. Since the condensed graph G' is a DAG, there cannot be an s-t path, traversing the same loops but in a different order. Thus all paths can be classified by the loops they traverse. Applying this insight recursively to all induced sub-loops of G, a path (respectively its weights) is uniquely defined by the set of induced sub-loops of G it traverses, yielding in total 2^{slbs} possible paths.

Theorem 2. *The modified algorithm reports a minimal number of longest paths.*

Proof. We want to prove the following claim by induction over $|\mathrm{iloops}(G)|$: For all $w_P \in \mathcal{D}(G, s, t)$, there exists an instantiation I such that under I (written: $w_P[I]$), for all weights $w_Q \in \mathcal{D}(G, s, t)$ such that $w_Q \neq w_P$ we have $w_P[I] > w_Q[I]$.

In the base case $|\mathrm{iloops}(G)| = 0$, G is a directed acyclic graph and the claim trivially holds, since $|\mathcal{D}(G, s, t)| = 1$. For the induction step, consider the condensed graph G' as constructed by the algorithm by replacing all loops $(L_i)_{1 \leq i \leq |\mathrm{loops}(G)|}$ in G by nodes $(c_i)_{1 \leq i \leq |\mathrm{loops}(G)|}$. We denote by $C := \{c_i \mid 1 \leq i \leq |\mathrm{loops}(G)|\}$ the set of all contraction nodes and for a path Q, we write $C_Q := \{c_i \in C \mid Q \text{ traverses } L_i\}$. Additionally, we write $w_{c_i}^Q$ for the weight polynomials corresponding to the loop L_i for a path Q.

In the following we will show that the claim holds for a particular instantiation: For all $c_i \in C \setminus C_P$ and for all $L \in \mathrm{iloops}(L_i)$ we set $I(\mathrm{b}(\mathscr{E}(L))) := 0$ if $\mathrm{b}(\mathscr{E}(L))$ is symbolic. For all $c_i \in C_P$ and for all $L \in \mathrm{iloops}(L_i)$ such that $\mathrm{b}(\mathscr{E}(L))$ is symbolic we will choose values that dependent on the weights in $\mathcal{D}(G, s, t) \setminus \{w_P\}$. For a moment, we will consider these weights in $\mathcal{D}(G, s, t)$ independent of each other. We will show

at the end of the proof, that we can do so as long as we only increase symbolic loop bounds occurring in w_P.

First note, that either w_P or w_Q must contain symbolic loop bound variables since otherwise both would just be constants and since w_P and w_Q are mutually non-dominating, this would lead to the contradiction that both are distinct longest path weights. If w_P is now a constant and w_Q not, then w_P must be larger than the constant term c in w_Q, because otherwise w_Q would dominate w_P and w_P would not have been reported by our algorithm. Since all symbolic loop bounds not occurring in w_P, i.e. all variables in this case, are set to 0, we have that $w_P = w_P[I] > w_Q[I] = c$. Thus, we can assume that w_P is not a constant. If now w_Q is a constant, any symbolic loop bound occurring in a term t in w_P can just be chosen such large, that t becomes larger than w_Q, which can be done, because – by construction of the weights – all coefficients in the weight polynomials must be positive. Thus, we can assume that both paths P and Q traverse loops of G.

Assume now, that the sets of loops traversed by P and by Q are disjoint. Because $w_Q[I]$ is a constant – recall that all variables corresponding to loops that are not traversed by P are set to 0 – and because w_P contains a variable b not contained in w_Q, we can make $w_P[I]$ larger than $w_Q[I]$ by choosing b large enough. Hence, we can assume now that there are loops that are traversed by both P and Q. Then, we can write P and Q as paths in the condensed graph G' in the following way. Let L_i be some loop that is traversed by P and by Q. Then $P = s \overset{P_1}{\rightsquigarrow} c_i \overset{P_2}{\rightsquigarrow} t$ and $Q = s \overset{Q_1}{\rightsquigarrow} c_i \overset{Q_2}{\rightsquigarrow} t$ for suitable sub-paths P_1 and P_2 of P, and for Q_1 and Q_2 of Q. Let $(v_1, v_2, \ldots, v_k, c_i, v_{k+1}, v_{k+2}, \ldots, v_l)$ (we assume here wlog. that $v_1 = s$ and $v_l = t$) be a topological ordering of the nodes of G' (note that this can be done, since G' is directed and acyclic). Consider now the two sub-graphs G_1 and G_2 of G, induced by the node sets $\{v_1, v_2, \ldots, v_k, \mathscr{E}(L_i)\}$ and $\{\mathscr{E}(L_i), v_{k+1}, v_{k+2}, \ldots, v_l\}$. Since any sub-path of a longest path must also be a longest path (with respect to validity), we know that $\{w_{P_1}, w_{Q_1}\} \subseteq \mathcal{D}(G_1, s, \mathscr{E}(L_i))$, $\{w_{P_2}, w_{Q_2}\} \subseteq \mathcal{D}(G_2, \mathscr{E}(L_i), t)$. Because G_1 and G_2 have less induced sub-loops than G, by induction hypothesis I can be chosen such that $w_{P_1}[I] \geq w_{Q_1}[I]$ and $w_{P_2}[I] \geq w_{Q_2}[I]$ – note that we have to allow equality here, since it is possible that the weights of the sub-paths are equal. Furthermore, we know for the same reason that $w^P_{c_i}[I] \geq w^Q_{c_i}[I]$, since after splitting the entry node of L_i, the loop also has less induced sub-loops than G. The crucial observation now is that at least one of these three inequalities has to be strict, because by induction hypothesis the inequalities can only be equalities if the corresponding weight polynomials are equal. So, if all of them would be equal, this would also mean that $w_P = w_Q$ which would be a contradiction to our assumption. Thus, $w_P[I] = w_{P_1}[I] + w^P_{L_i}[I] + w_{P_2}[I] > w_{Q_1}[I] + w^Q_{L_i}[I] + w_{Q_2}[I] = w_Q$. What is left to show is that we can indeed consider the weights in $\mathcal{D}(G, s, t)$ independently. We will show that given an instantiation I as constructed above for the two weights w_P and w_Q, for all instantiations I' with the properties that $I'(s) \geq I(s)$ for all symbolic loop bounds s and $I'(s) = 0$ if s is not contained in w_P, we have $w_P[I'] > w_Q[I']$. We will prove the claim by structural induction over G. If G is a directed acyclic graph, nothing has to be shown. Thus, we can assume that G contains loops. Consider the paths P and Q as paths in the condensed graph G'. By the construction of I we know that

$w^P_{c_i}[I] > w^Q_{c_i}[I]$ for $c_i \in C_P \cap C_Q$ if $w^P_{c_i} \neq w^Q_{c_i}$ and thus, by induction hypothesis, $w^P_{c_i}[I'] > w^Q_{c_i}[I']$. We now have

$$\begin{aligned} w_P[I'] &= \sum_{e\in P} w(e) + \sum_{c_i\in C_P\cap C_Q} I'(L_i) w^P_{c_i}[I'] + \sum_{c_i\in C_P\setminus C_Q} I'(L_i) w^P_{c_i}[I'] \\ &\overset{(I)}{>} \sum_{e\in P} w(e) + \sum_{c_i\in C_P\cap C_Q} I'(L_i) w^Q_{c_i}[I'] + \sum_{c_i\in C_P\setminus C_Q} I'(L_i) w^P_{c_i}[I'] \\ &\overset{(II)}{\geq} \sum_{e\in Q} w(e) + \sum_{c_i\in C_P\cap C_Q} I'(L_i) w^Q_{c_i}[I'] + \sum_{c_i\in C_Q\setminus C_P} I'(L_i) w^Q_{c_i}[I'] \;=\; w_Q[I'] \end{aligned}$$

where inequality (I) follows by induction hypothesis (to simplify the discussion we assume wlog. that there is at least one common loop that is traversed differently by P and Q). Inequality (II) clearly holds if

$$\sum_{e\in P} w(e) + \sum_{c_i\in C_P\setminus C_Q} I'(L_i) w^P_{c_i}[I'] \geq \sum_{e\in Q} w(e) + \sum_{c_i\in C_Q\setminus C_P} I'(L_i) w^Q_{c_i}[I']$$

But this must be true, for two reasons: First, $\sum_{c_i\in C_Q\setminus C_P} I'(L_i) w^Q_{c_i}[I'] = \sum_{c_i\in C_Q\setminus C_P} I(L_i) w^Q_{c_i}[I]$ because all variables not contained in w_P are set to 0 in I and in I' and second, $\sum_{c_i\in C_P\setminus C_Q} I'(L_i) w^P_{c_i}[I'] \geq \sum_{c_i\in C_P\setminus C_Q} I(L_i) w^P_{c_i}[I]$ by induction hypothesis and the property that $I'(L_i) \geq I(L_i)$.

4 Conclusion

We presented an algorithm to compute longest path in control flow graphs. In earlier work [2], we performed experiments that demonstrate that our algorithm is the first algorithm that allows us to analyze the worst case execution times of programs with several symbolic loop bounds. In this paper, we give the theoretical foundation. We analyzed running time and the output size of our algorithm for the case of programs avoiding goto-constructs. We tailored the analysis of the output size, if one additionally avoids break and continue statements. This explains the observed small output sizes that are in contrast to the worst case doubly exponential output size that is possible for general control flow graphs.

In future work, we plan to investigate on the difference between the upper and the lower bound on the minimal output size and the difference of the output size of our algorithm and the minimal possible output size for programs avoiding goto-constructs but allowing break and continue statements.

References

1. Althaus, E., Altmeyer, S., Naujoks, R.: A new combinatorial approach to parametric path analysis. Reports of SFB/TR 14 AVACS 58, SFB/TR 14 AVACS (June 2010), `http://www.avacs.org`, ISSN: 1860-9821
2. Althaus, E., Altmeyer, S., Naujoks, R.: Precise and efficient parametric path analysis. In: Proceedings of the ACM SIGPLAN/SIGBED 2011 Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES, pp. 141–150 (2011)

3. Altmeyer, S., Hümbert, C., Lisper, B., Wilhelm, R.: Parametric timing analyis for complex architectures. In: Procedeedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2008), Kaohsiung, Taiwan, pp. 367–376. IEEE Computer Society, Los Alamitos (2008)
4. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press, Princeton (1963)
5. Feautrier, P.: The parametric integer programming's home, `http://www.piplib.org`
6. Gomory, R.E.: An algorithm for integer solutions to linear programming. In: Graves, R.L., Wolfe, P. (eds.) Recent Advances in Mathematical Programming, pp. 269–302. McGraw-Hill, New York (1969)
7. Holsti, N., Gustafsson, J., Bernat, G., Ballabriga, C., Bonenfant, A., Bourgade, R., Cassé, H., Cordes, D., Kadlec, A., Kirner, R., Knoop, J., Lokuciejewski, P., Merriam, N., de Michiel, M., Prantl, A., Rieder, B., Rochange, C., Sainrat, P., Schordan, M.: Wcet 2008 – report from the tool challenge 2008 – 8th intl. workshop on worst-case execution time (wcet) analysis. In: Kirner, R. (ed.) 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Dagstuhl, Germany, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2008); also published in print by Austrian Computer Society (OCG) under ISBN 978-3-85403-237-3
8. Li, Y.-T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: DAC 1995: Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference, pp. 456–461. ACM, New York (1995)
9. Lisper, B.: Fully automatic, parametric worst-case execution time analysis. In: Third Internation Workshop on Worst-Case Execution Time Analysis, pp. 77–80 (July 2003)
10. Tan, L.: The worst case execution time tool challenge 2006: Technical report for the external test. In: Proc. 2nd International Symposium on Leveraging Applications of Formal Methods, ISOLA 2006 (2006)
11. Theiling, H.: ILP-based Interprocedural Path Analysis. In: Proceedings of the Workshop on Embedded Software, Grenoble, France (October 2002)
12. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. 7(3), 1–53 (2008)

A Appendix

Lemma 3. *For any problem instance* $I = (G, s, t)$ $|\mathcal{D}(I)| \leq 2^{2^{\mathrm{lbs}(G)}}$.

Proof. Let P be a path reported by our algorithm. We recursively define the loop-pattern $\mathcal{L}(P, G)$ of a path P in G as follows. If G is a DAG, the loop-pattern of any path is empty. To define a loop-pattern in the general case, we shortly recall some facts of the algorithm. We compute P as a longest path in the condensed graph (for which we have contracted each loop of G into a single node), which is known to be an acyclic graph, since the loops are the strongly connected components of G. Let $L_1, \ldots, L_k$ be the subloops of G that are entered by P. Each loop L_i is entered exactly once (as the condensed graph is a DAG). Within the loop, P traverses a unique subpath P_i' for $b_{L_i} - 1$ times and then a path P_i'' to a portal node. The loop pattern of P is then defined as the sequence

$$\begin{aligned}\mathcal{L}(P) := (&L_1(\mathcal{L}(P_1', L_1'), \mathcal{L}(P_1'', L_1')), L_2(\mathcal{L}(P_2', L_2'), \mathcal{L}(P_2'', L_2')), \ldots,\\ &L_k(\mathcal{L}(P_k', L_k'), \mathcal{L}(P_k'', L_K'))),\end{aligned}$$

where L_i' is the graph obtained by splitting $\mathscr{E}(L_i)$.

We proof the following claims:

1. Any two paths with the same loop-pattern (not necessarily with same source and target node) computed by the algorithm have the same cost up to a constant term.
2. Any two $s-t$-paths reported by the algorithm have different loop-patterns.
3. Let $T(\mathrm{lbs}(G))$ be the maximum possible number of loop patterns of a graph G, then T can be bounded by the recurrence $T(0) = 1, T(\mathrm{lbs}(G)) \leq T(\mathrm{lbs}(G) - 1)^2 + 1$.

We will prove the points in turn. We proof the first point by structural induction over G. If G is a DAG, all path weights are constants (as there are no symbolic loop-bounds). Hence, there is nothing to show. Now consider two paths P and Q in a graph G constructed by our algorithm with $\mathcal{L}(P, G) = \mathcal{L}(Q, G)$. Let $L_1, \ldots, L_k$ be the subloops of G contained in $\mathcal{L}(P, G)$. Decompose P into $(\bar{P}_0, (P_1')^{b_{L_1}-1}, P_1'', \bar{P}_1, \ldots, (P_k')^{b_{L_k}-1}, P_k'', \bar{P}_k)$, where $\bar{P}_i$ is the path from the portal of L_i used by P (respectively from the source of P if $i = 0$) to $\mathscr{E}(L_{i+1})$ (respectively to the target of P for $i = k$), P_i' is the path used for cycling within L_i and P_i'' if the path from the entering node of L_i to the portal-node used by P. Analogously decompose Q. The cost of P can then be written as

$$w(\bar{P}_0) + \sum_{i=1}^{k} [(b_L - 1)w(P_i') + w(P_i'') + w(\bar{P}_i)],$$

the cost of Q as

$$w(\bar{Q}_0) + \sum_{i=1}^{k} [(b_L - 1)w(Q_i') + w(Q_i'') + w(\bar{Q}_i)].$$

The costs of $\bar{P}_i$ and $\bar{Q}_i$ are some constants. By induction hypothesis, the costs of P_i' and Q_i' only differ by a constant. The same holds for the cost of P_i'' and Q_i''. Hence the difference of $w(P)$ and $w(Q)$ is a sum of constants and thus constant.

The second point immediately follows from the first, as our algorithm won't report two paths whose weights only differ in a constant.

For the third point, we argue as follows: Let $L_1, \ldots, L_k$ be the subloops of G given in topologial order. Each loop pattern can be constructed by chosing for each subloop L_i either that it is not entered, or we use some loop pattern of L_i for the cycling path and one loop pattern for the path to the portal. Hence, we get $T(\mathrm{lbs}(G)) \leq \prod_i T(\mathrm{lbs}(L_i'))^2 + 1$. Notice that $\sum_i \mathrm{lbs}(L_i) = \mathrm{lbs}(G)$ and $\mathrm{lbs}(L_i') = \mathrm{lbs}(L_i) - 1$ and hence $\sum_i \mathrm{lbs}(L_i') = \mathrm{lbs}(G) - k$. A simple calculation shows that $T(\mathrm{lbs}(G))$ is maximized, if each induced subloop of G containts exactly one subloop. Hence we get $T(\mathrm{lbs}(G)) \leq T(\mathrm{lbs}(G) - 1)^2 + 1$.

Note that if L_i as a numeric loop-bound, the length of a path that does not enter L_i and the length of a path that enteres L_i but no subloop of L_i differ only in a constant. Hence, in this case we can drop the addition of one in the recursive formula. Finally for $T(\mathrm{lbs}(G)) = T(\mathrm{lbs}(G) - 1)^2 + 1, T(0) = 1$ holds $T(\mathrm{lbs}(G)) \leq 2^{2^{\mathrm{lbs}(G)}}$ and for $T'(\mathrm{lbs}(G)) = T'(\mathrm{lbs}(G) - 1)^2, T'(1) = 2$ holds $T'(\mathrm{lbs}(G)) \geq 2^{2^{\mathrm{lbs}(G)-1}}$.

Lemma 4. *There exists a problem instance $I = (G, s, t)$, such that $|\mathcal{D}(I)| = 2^{2^{\mathrm{lbs}(G)-1}}$ and $\mathrm{slbs}(G) = 1$.*

Proof. We will first prove a slightly different claim, namely we proof the bound using symbolic loop-bounds for all loops. More precisely, we show that there exists a problem instance I such that

- a) $\mathcal{D}(I) = 2^{2^{\mathrm{slbs}(G)-1}}$
- b) there is an element in $\mathcal{D}(I)$ containing a positive constant
- c) there is an element in $\mathcal{D}(I)$ containing the constant 0

where x is the number of symbolic loop-bounds.

We construct a graph G in the following way: we start with the left graph G_l in Figure 3 and repeatedly replace the selfloop of G by the loop of G_l. Note, that after k iterations, the resulting graph consists of $k + 2$ induced subloops. We will show the claim by induction over the recursive structure of G, i.e. over the number $\mathrm{slbs}(G)$ of induced subloops of G.

For the base case ($\mathrm{slbs}(G) = 2$), in which G corresponds to the graph G_l, it is easy to verify that

$$\mathcal{D}(v_0, v_2) = \{b(v_0), b(v_0)b(v_1), -1 + b(v_0) + b(v_1), 1 - b(v_1) + b(v_0)b(v_1)\}$$

and that one of these weight polynomials contains a positive constant.

Now let us consider the induction step. Recall that we want to compute $\mathcal{D}(v_0, v_2)$. In the first step of our algorithm, the node v_0 is split into two nodes v_{in} and v_{out} and recursively the sets $\mathcal{D}(v_1, v_1)$, $\mathcal{D}(v_{out}, v_{in})$ and $\mathcal{D}(v_{out}, v_2)$ are computed. Then, the set $\mathrm{lps}(v_0, v_2)$ is computed as the set of polynomials given by $(\mathrm{b}(v_0) - 1) \cdot l_{ee} + l_{ep}$ for all $l_{ee} \in \mathcal{D}(v_{out}, v_{in})$ and for all $l_{ep} \in \mathcal{D}(v_{out}, v_2)$.

By the way, we have chosen the edge weights, we have that $\mathcal{D}(v_{out}, v_{in}) = \mathcal{D}(v_{out}, v_2) = \mathcal{D}(G, v_1, v_1)$. In particular, note that the constant polynomial 1 is not in contained in $\mathcal{D}(v_{out}, v_{in})$, since by induction hypothesis, $\mathcal{D}(v_1, v_1)$ contains a polynomial with a positive constant (which must be greater or equal to 1), dominating 1.

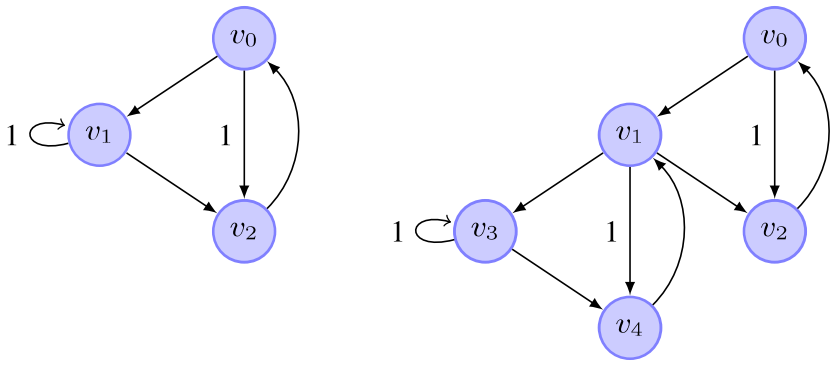


Fig. 3. Suppose the edge weights are 0 if not stated otherwise, then the longest path weights from v_0 to v_2 constructed by our algorithm for the left graph are $b(v_0)$, $b(v_0)b(v_1)$, $-1+b(v_0)+b(v_1)$ and $1-b(v_1)+b(v_0)b(v_1)$. The right graph was constructed from the left one by replacing the selfloop $\{v_1\}$ by the loop $\{v_0, v_1, v_2\}$. For the right graph, the algorithm computes 16 non-dominant longest path weights. In general, graphs obtained by repeatedly replacing the selfloop by the loop of the left graph, yield $2^{2^{\mathrm{slbs}(G)-1}}$ non-dominant longest path weights and furthermore, one can show that there is a longest path weight that consists of $2^{\mathrm{slbs}(G)}-1$ terms where $\mathrm{slbs}(G)$ is the number of symbolic loop-bounds.

This in turn, means that $|\mathrm{lps}(v_0,v_2)| = |\mathcal{D}(v_1,v_1)|^2$. We now have to show, that the elements in $\mathrm{lps}(v_0,v_2)$ are pairwise non-dominating. Let $l_1, l_2 \in \mathrm{lps}(v_0,v_2)$, given as $l_1 = (\mathrm{b}(v_0)-1)\cdot l_{ee} + l_{ep}$ and $l_2 = (\mathrm{b}(v_0)-1)\cdot l'_{ee} + l'_{ep}$ for weights $l_{ee}, l'_{ee} \in \mathcal{D}(v_{out}, v_{in})$ and $l_{ep}, l'_{ep} \in \mathcal{D}(v_{out}, v_{in})$, l_1. We distinguish two cases. In the first one, we assume that $l_{ee} \neq l'_{ee}$. Since by induction hypothesis, l_{ee} and l'_{ee} are pairwise non-dominant, the terms in l_1 and l_2 containing variable $\mathrm{b}(v_0)$, namely $\mathrm{b}(v_0)\cdot l_{ee}$ and $\mathrm{b}(v_0)\cdot l'_{ee}$ - and thus also l_1 and l_2 - must be pairwise non-dominant. Now we assume that $l_{ee} = l'_{ee}$. In this case, the terms, not containing the variable $\mathrm{b}(v_0)$, are $l_{ep} - l_{ee}$ and $l'_{ep} - l_{ee}$. Since by induction hypothesis l_{ee} and l'_{ee} are pairwise non-dominant, l_1 and l_2 must also be pairwise non-dominant.

Thus, $|\mathcal{D}(v_0,v_2)| = |\mathcal{D}(v_1,v_1)|^2$. Since by induction hypothesis $|\mathcal{D}(v_1,v_1)| = 2^{2^{x-2}}$, $|\mathcal{D}(v_0,v_2)| = 2^{2^{x-1}}$, which establishes the first part of the claim. For the second part, note that by induction hypothesis, there is a weight in $l_1 \in \mathcal{D}(v_1,v_1)$ with a positive constant and there is a weight in $l_2 \in \mathcal{D}(v_1,v_1)$ with zero constant. Thus the weight $l \in \mathcal{D}(v_0,v_2)$ with $l = (\mathrm{b}(v_0)-1)\cdot l_2 + l_1 = \mathrm{b}(v_0)\cdot l_2 + l_1 - l_2$ has again a postive constant as a term. On the other hand, the weight $l \in \mathcal{D}(v_0,v_2)$ with $l = (\mathrm{b}(v_0)-1)\cdot l_2 + l_2 = \mathrm{b}(v_0)-1)\cdot l_2$ has a zero as constant term.

So far, we have assumed that all loop-bounds are symbolic. Now we will examine an instance in which all but one loop-bounds are numeric. For this, we use the same graph G as constructed above, but we assume that only the bound for the selfloop is symbolic. Then we can show by the very same induction as above, that by choosing the right numeric values for the loop-bounds, $|\mathcal{D}(v_0,v_2)| = 2^{2^{\mathrm{iloops}(G)-1}}$. For the base case, we choose $b(v_0) := 3$ and obtain $\mathcal{D}(v_0,v_2) = \{3, 2+\mathrm{b}(v_1), 1+2\mathrm{b}(v_1), 3\mathrm{b}(v_1)\}$ which clearly satisfies the induction properties.

The only difference occurs now in the induction step, when arguing, that no two path weights l_1, l_2 exist, such that the first one dominates the other. Since there is just

one symbolic loop-bound, all path weight polynomials consist of only two terms, i.e. $\mathcal{D}(v_1, v_1)$ consists of polynomials $c_i + c'_i \cdot a$ where a is the symbolic loop-bound in G. It is easy to see that these polynomials are non-dominated if and only if they can be ordered in such a way, that the constants c_i are appear in increasing order while the c'_i appear in decreasing order.

The idea of proof is now to show, that the numeric bound $\mathrm{b}(v_0)$ can recursively be chosen in such a way, that the constructed weights in $\mathcal{D}(v_0, v_2)$ can be ordered in the same way. To see this, suppose we are given two weights in $\mathcal{D}(v_0, v_2)$, namely $p_1 := (\mathrm{b}(v_0)-1)\cdot d_1 + d'_1$ and $p_2 := (\mathrm{b}(v_0)-1)\cdot d_2 + d'_2$ for $d_1, d_2, d'_1, d'_2 \in \mathcal{D}(v_0, v_2)$. Clearly by choosing $\mathrm{b}(v_0)$ big enough, it is possible to put p_1 and p_2 into the same relative order as d_1 and d_2. We will now show, that there is such a value for $\mathrm{b}(v_0)$ for all pairs of weights in $\mathcal{D}(v_0, v_2)$ that is not too big.

Consider again the pair p_1, p_2. Let us denote by $c(p)$ the constant term of such a weight p and by $v(p)$ the coefficient of the variable term. Without loss of generality, let us restrict our discussion to the case $c(d_1) \geq c(d_2)$. It is easy to verify that for $\mathrm{b}(v_0) := \left\lceil \max\left\{ \frac{c(d'_2)-c(d'_1)}{c(d_1)-c(d_2)}, \frac{v(d'_1)-v(d'_2)}{v(d_2)-v(d_1)} \right\} \right\rceil + 1$ we have $c(p_1) \geq c(p_2)$ and $v(p_1) \leq v(p_2)$.

Choosing $\mathrm{b}(v_0)$ as the maximum of all possible choices of p_1,$p_2 \in \mathcal{D}(v_0, v_2)$ we can establish the same relative order as for the d_i, completing the proof.

Lemma 5. *There exists a problem instance I, such that any correct algorithm must report $2^{\mathrm{slbs}(G)}$ longest paths.*

Proof. Consider the graph G_f in Figure 2. By repeated concatenation of the subgraph of G_f induced by the nodes $\{v_0, s_1, v_1\}$, we obtain a weighted graph $G = (V, E, w)$ that consists of nodes $V = \{v_0, \ldots, v_{\mathrm{slbs}(G)}, s_0, \ldots s_{\mathrm{slbs}(G)-1}\}$, edges

$$E = \{(v_i, s_i), (s_i, v_i), (s_i, s_i), (v_i, v_{i+1}) \mid i \in \{0, \ldots, \mathrm{slbs}(G) - 1\} \cup \{(s_{\mathrm{slbs}(G)-1}, v_{\mathrm{slbs}(G)})\}\}$$

and of edge weights as given in the graph G_f.

Then there are exactly $2^{\mathrm{slbs}(G)}$ different paths from v_0 to $v_{\mathrm{slbs}(G)}$, namely one for each choice of bypassing a selfloop $\{s_i\}$ via the edge (v_i, v_{i+1}) or not. For these paths we have the set of corresponding weights

$$\left\{ \sum_{p_i \in p} p_i \cdot \mathrm{b}(s_i) + 2 \cdot \sum_{p_i \in p} (1 - p_i) \;\mid\; p \in \{0, 1\}^{\mathrm{slbs}(G)} \right\}$$

The claim now is, that for each weight in this set, there is an instantiation I of its symbolic loop-bounds, such that this weight dominates all other weights. To see this, consider a weight w. For each loop-bound variable s_i contained in w, set $I(\mathrm{b}(s_i)) := 2\mathrm{slbs}(G) + 1$ and for all other bounds to 0. Now consider any other weight w' and let n (n') be the number of variables in w (in w') and k be the number of variables that w and w' share. Then $w[I] = n(2\mathrm{slbs}(G) + 1) + 2(\mathrm{slbs}(G) - n)$ and $w'[I] = k(2\mathrm{slbs}(G) + 1) + 2(\mathrm{slbs}(G) - n')$. If now $k = n$, then there must be a variable in w' which is not in w, since $w \neq w'$. Thus, $n' > n$ and therefore $w[I] = n(2\mathrm{slbs}(G)+1)+2(\mathrm{slbs}(G)-n) > n(2\mathrm{slbs}(G) + 1) + 2(\mathrm{slbs}(G) - n') = w'[I]$. Since $k \leq n$, let us now assume that $k < n$. Then $w[I] - w'[I] = (n - k)(2\mathrm{slbs}(G) + 1) + 2(n' - n)$ can only be negative

if $n' < n$, but on the other hand $n' - n \geq -\mathrm{slbs}(G)$, which implies that in this case $w[I] - w'[I] \geq (n-k)(2\mathrm{slbs}(G)+1) - 2\mathrm{slbs}(G) = 2\mathrm{slbs}(G)(n-k-1)+n-k > 0$. Hence any algorithm has to report w.

Thus any correct algorithm must report all the $2^{\mathrm{slbs}(G)}$ paths.

Lemma 6. *There exists a problem instance I, such that $\mathcal{D}(I)$ contains a weight with $2^{\mathrm{slbs}(G)} - 1$ terms with non-zero coefficients.*

Proof. Consider the same graph construction as in Lemma 4. We again prove a slightly stricter claim, namely that there is

- a weight polynomial in $\mathcal{D}(G, v_0, v_2)$ such that all terms have non-zero coefficients except the term consisting of all loop-bound variables
- a weight polynomial in $\mathcal{D}(G, v_0, v_2)$ such that all terms have zero coefficients except the term consisting of all loop-bound variables.

We again proof the claim by induction. As stated in the proof of this lemma, in the base case ($\mathrm{slbs}(G) = 2$), the set of computed path weights is $\mathcal{D}(v_0, v_2) = \{b(v_0), b(v_0)b(v_1), -1 + b(v_0) + b(v_1), 1 - b(v_1) + b(v_0)b(v_1)\}$. Clearly the claim holds in this case. For the induction step, consider the weights $l, l' \in \mathcal{D}(v_1, v_1)$ such that l corresponds to the weight in the first part of the claim and l' corresponds to the weight in the second part of the claim. Recall from the discussion in the proof of Lemma 4, that $\mathcal{D}(v_0, v_2)$ consists of weights build by evaluating the expression $(\mathrm{b}(v_0) - 1) \cdot l_1 + l_2$ for l_1,$l_2 \in \mathcal{D}(v_1, v_1)$. Since $l \in \mathcal{D}(v_1, v_1)$, $(\mathrm{b}(v_0) - 1) \cdot l + l = \mathrm{b}(v_0) \cdot l$ is in $\mathcal{D}(v_0, v_2)$ and thus the first part of the claim also holds for G. But also the weight $(\mathrm{b}(v_0) - 1) \cdot l + l' = \mathrm{b}(v_0) \cdot l + (l' - l)$ is in $\mathcal{D}(v_0, v_2)$. Since l and l' have distinct terms with non-zero coefficients and by induction hypothesis, l has $2^{\mathrm{slbs}(G)-1} - 1$ terms with non-zero coefficients, the number of terms in $(\mathrm{b}(v_0) - 1) \cdot l + l'$ with non-zero coefficients must be $2(2^{\mathrm{slbs}(G)-1} - 1) + 1 = 2^{\mathrm{slbs}(G)} - 1$, which finishes the proof.

Lemma 7. *There is a problem instance $I = (G, s, t)$, such that the minimal output size is 2 but $|\mathcal{D}(I)| = 2^{2^{\mathrm{lbs}(G)-1}}$.*

Proof. Reconsider again the example given in Figure 3 and the graph G as constructed in the proof of Lemma 4. We have argued in this proof, that the set $\mathrm{lps}(v_0, v_2)$ of path weights consists of all weights $(\mathrm{b}(v_0) - 1) \cdot l_{ee} + l_{ep}$ for $l_{ee}, l_{ep} \in \mathcal{D}(v_1, v_1)$, since $\mathcal{D}(v_{out}, v_{in}) = \mathcal{D}(v_{out}, v_2) = \mathcal{D}(G, v_1, v_1)$. But one could do better, since it can be assumed wlog. that a longest path from v_0 to v_2 just corresponds to $\mathrm{b}(v_0)$ times a path in $\mathcal{D}(v_1, v_1)$. Thus, instead creating a set $\mathrm{lps}(v_0, v_2)$ of cardinality $|\mathcal{D}(v_1, v_1)|^2$, we create one of cardinality $|\mathcal{D}(v_1, v_1)|$. But this leads by an inductive argument to a set of weights of the base case, namely 4 path weights. You could also apply this idea to the base case and get just the two path weights $\mathrm{b}(v_0)$ and $\mathrm{b}(v_0)\mathrm{b}(v_1)$, but then the induction step in the proof of Lemma 4 would not work out in the same way, as there would be no path weight in $\mathcal{D}(v_1, v_1)$ with a positive constant (even though, it is possible to modify G slightly to make it work). So, we end up with four (respectively 2) longest path weights which still is in stark contrast to the set of $2^{2^{\mathrm{lbs}(G)-1}}$ weights, constructed by our algorithm.